CHAPTER 3
GETTING TO KNOW YOU, ENUMERATION

Let us assume that the PC host meets all of the requirements described in Chapter 1, is running a USB-aware operating system, and has an available USB port. This port could be on the PC host itself (an embedded hub) or could be on an external hub. Now we have a new USB I/O device that we want to add to this running system. What actually happens between the host, the hub, and the device to deliver the many USB features?

After understanding what the PC host is doing, we learn the responsibilities of a general I/O device. All devices describe themselves using a collection of specially formatted bytes called descriptor tables. We start by looking inside the simplest example and then expand the discussion to cover the general case. We then derive the minimum hardware requirements for an I/O device. I will keep this chapter as general as possible so that the discussion will not be constrained by specific product implementations. We shall look at real products in the next chapter.

There are many “chicken-vs.-egg” situations in this section, so I’ll need to defer some technical discussions to keep the flow of the concepts moving forward.

DEVICE DETECTION

Figure 3-1 shows details about the USB cable. The cable has four wires: two power wires for Vbus and Gnd and two signal wires for D+ and D–. The cable end that attaches to the hub has a Series A connector, and the cable end that attaches to the new device is either connected directly (no connector) or has a Series B connector. Both connectors have longer power and ground connector pins to ensure that the device has good voltages before signals are applied.

The hub port supplies Vbus and Gnd. The current limiter will initially prevent more than 100 mA from being drawn, even instantaneously, from the hub. If excess current is drawn, then the hub informs the host software of this error (see "Enumeration steps," step 6), an error message is displayed on the PC screen, and the device is **not** configured.

Because we haven't plugged in the I/O device yet, it is in the **unattached** state.

In Figure 3-1, note the two biasing resistors in the hub; they ensure that D+ and D– are low when no device is plugged in. There is a single biasing resistor on the device that is attached to either D+ or D–. When the USB cable is plugged in, the biasing resistor causes D+ or D– to rise above ground, and this changed voltage difference is recognized by the hub. We have detected a cable being plugged in! By convention, if the device-biasing resistor is connected to D+, we are informing the hub that this device is full speed (12 Mbps), while a biasing resistor on D– indicates a low speed (1.5 Mbps) device. Simple and effective!

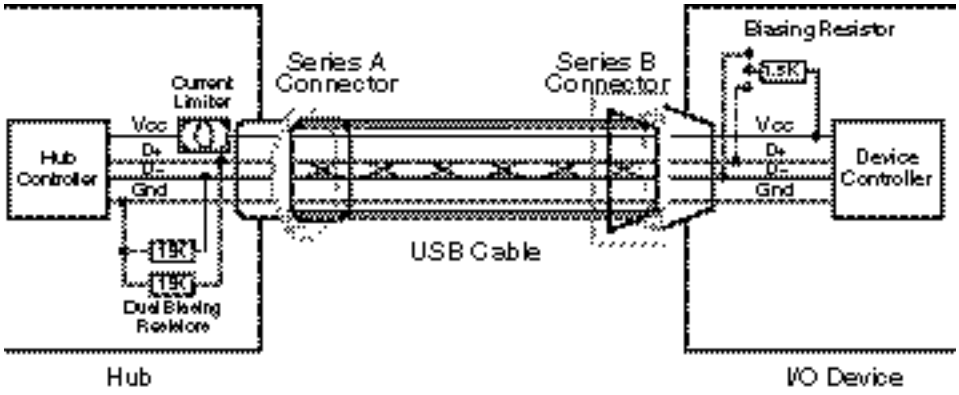


Figure 3-1. USB cable connection details

Note that a high-speed hub or a high-speed device has not been identified yet. A high-speed device is first detected as a full speed device and will alert the hub of its high-speed capabilities a little later in the sequence.

The hub detected the presence of an I/O device by the change in DC level caused by the I/O device biasing resistor. Imagine, for a moment, that this biasing resistor was connected to Vbus via a transistor switch. Turn the transistor **ON** and the hub recognizes the I/O device being attached. Turn the transistor **OFF** and the hub recognizes this as the I/O device being disconnected. If the device controller had the capability to turn this transistor **ON** and **OFF** it could programmatically connect and disconnect itself from the PC host. This technique is very useful if the I/O device requires extensive initialization before being connected to the PC host. The technique is used by many commercial I/O devices that re-program themselves as a different device and disconnect-reconnect with a new personality.

The I/O device is now in the **attached** state, and, because this hub is already configured and operational, the device moves to its **powered** state.

The hub updates a STATUS_CHANGE register following detection of the new device and then waits to be told what to do.

The PC host controls the enumeration phase and, during this phase, sends requests to two devices. The hub that identified the newly attached device will receive many requests for action, and the newly attached I/O device will also receive requests. If there are any other hubs between this hub and the root hub, they will not take part in this process. They will repeat the signals on the USB, because that is one of their roles, but because they are not being addressed by the PC host software during this process, they may be ignored.

The PC host software regularly polls all connected hubs looking for work to do. In most cases a hub has nothing to report so will NAK this interrupt transfer. But **this** time the hub responds with the STATUS_CHANGE data indicating which port has had a change in status—the PC host-based enumeration process has begun!

ENUMERATION STEPS

In the following description the PC host is initiating all requests. I have used a **ToHub:** prefix if the addressed device is the hub, or a **ToIO:** prefix if the addressed device is the newly attached I/O device.

1. **ToHub:Get_Port_Status**: Host discovers newly attached device.
2. **ToHub:Clear_Port_Feature(C_PORT_CONNECTION)**: Clears the flag in the STATUS_CHANGE register that started this process.
3. **ToHub:Set_Port_Feature(PORT_RESET)**: The hub responds by sending a reset to the I/O device. The hub maintains this reset for a minimum of 10 milliseconds. It then updates the RESET_CHANGE bit in its PORT_CHANGE register and enables the port for USB traffic by setting the PORT_ENABLE bit in the PORT_STATUS register. The PORT_CHANGE register update causes an update to the STATUS_CHANGE register. The PC host will notice this on its next scheduled poll.
4. During this reset time, high-speed devices implement a handshake sequence, (described in the next section) and, if successful, change this USB segment speed to 480-Mb/s.
5. **ToHub:Get_Port_Status**: The PC host discovers that the reset is complete.
6. **ToHub:Clear_Port_Feature(C_PORT_RESET)**: Clears the flag in the STATUS_CHANGE register. At this time the I/O device has power and has been reset; therefore, it is in its **default** state. The device will now respond to PC host requests to device address 0 (the default address). If *another* device were attached at this same instant in time, the PC host software will defer servicing it until the enumeration sequence on the *current* I/O device is completed. In other words, there will be only one device that responds to device address 0 at any one time.
7. **ToIO:Get_Device_Descriptor**: The PC host makes its first move to discover what kind of device has been attached. The device responds with a device descriptor (Figure 3-2). This data structure is described in the next section.
8. **ToIO:Set_Address**: The PC host allocates a device address to the newly attached I/O device. All subsequent requests are sent to this new device address. The device is now in its **addressed** state.
9. **ToIO:Get_Device_Descriptor**: The PC host repeats this request to the new address. It should get the same response shown in Figure 3-2. Any different response, or a device timeout, indicates an error condition that the PC host software must deal with.

10. **ToIO:Get_Configuration_Descriptor**: The device driver starts collecting information about the device, its interfaces, and its endpoints. In a feature-rich I/O device, the configuration can be quite extensive (for a preview, see Figure 3-15). For now, we will review the simpler example in Figure 3-3. All of the parameters will be explained later in this chapter. After some processing that may (hopefully not) require some user interaction, the host software will move to the next step.
11. **Select Device Driver**: From the PC host's perspective, it needs to determine which device driver is needed to support this newly attached I/O device. The ***Choosing a device driver*** algorithm is covered later in this chapter. If the selected device driver is not currently in memory, it is loaded now. If the device has multiple configurations, then the driver typically reads them all and uses some algorithm to choose one of them (often the first one!).
12. **ToIO:Set_Configuration**: The device is now configured and operational, so it moves into its **configured** state.

High-Speed Handshake

The reset signal driven by the hub is a single-ended zero (both D+ and D- low) so it is easy for the I/O device to detect the start of this signal. A high-speed device initially declared itself as a full-speed device by connecting a biasing resistor to the D+ line. It is now time to declare its high speed capability and it does this by signaling a "chirp" soon after detecting the reset signal. A chirp is a 480-Mb/s K state that is minimally 1 msec and maximally 7 msec.

A full-speed hub will ignore this chirp and this USB segment will remain at 12Mb/s.

A high-speed hub will recognize this chirp as a request from a high-speed device to increase the speed of this USB segment. A high-speed hub will acknowledge this request by sending a sequence of alternating Chirp K and Chirp J signals as soon as the I/O device Chirp K is completed but before the 10 msec reset sequence has completed. The hub will switch to high-speed on this USB segment once the reset signaling is completed.

If the full-speed I/O device does not receive this alternating Chirp K, Chirp J response it does not change its speed (it must be attached to a full-speed hub which doesn't know about Chirps and can only run at 12Mb/s anyway!)

If the full-speed I/O device does receive the high-speed hub handshake of Chirp K-J-K-J-K then it will switch into high-speed mode. This mode change involves disconnecting the biasing resistor so that the high-speed line is symmetrically

terminated and has a default state of a single-ended zero (SE0=D+ and D- low). High-speed drivers and terminators will also be enabled, and the I/O device enters its **high-speed default** state.

This high-speed handshake is a little involved but has the enormous benefit of operating correctly in a mixed high-speed/full-speed environment. It is fully upwards compatible with full/low-speed products and allowed the high-speed capability to be added to USB with no change in the USB usage model.

The remainder of this chapter discusses the blocks of identification information that an I/O device must supply and all of these operations are independent of the speed of the individual USB segments.

Device Descriptor

Figure 3-2 shows the device descriptor that is returned following the PC host's Get_Device_Descriptor request (step 7). It is a collection of 18 bytes which are formatted with standard, device-specific information. Most of the fields are 8 bit values but some are 16 bit values. We'll investigate each parameter but will use default values during this early discussion.

This descriptor information is similar to the configuration information supported by the PCI bus specification and the PnP information optionally supported by the ISA bus specification. The goal of the information is the same – provide the operating system software with enough information that will allow the automatic configuration and installation of this new I/O device into the PC system environment.

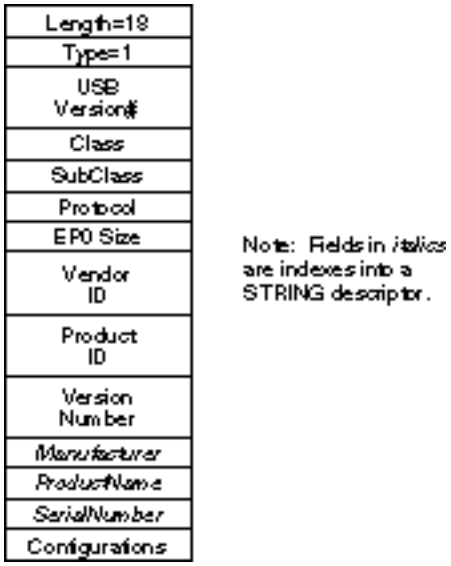


Figure 3-2. Device response to GET_DEVICE_DESCRIPTOR

The first three values, **Length** of this descriptor, **Type** of this descriptor, and **Version** of the USB Specification that we are compliant to, are easy to fill in (18, 1, and 200H, respectively).

<< HELP! What does the OS do different for USB version 1.0, 1.1 and 2.0?>>

The **Class, Subclass,** and **Protocol** need some explanation.

Classes were created within the USB Specification to make it easier to write PC host software. In some cases, as we shall see, we do not need to supply **any** host software. Classes are somewhat confusing and appear unnecessary to the hardware designer, but because much of USB revolves around them, we will have to learn this subject! So what are the benefits of classes to the hardware designer?

The host operating system groups the diverse world of I/O devices into classes: I/O devices with similar characteristics are grouped together, and a generic device driver is provided for this class. Some examples include “Human-Interface-Device,” “Audio,” “Mass Storage,” and “Power Supplies.”

A subclass is a finer granularity grouping within a class. Protocol is used in some class definitions.

So, if we can describe an I/O device within one of the predefined classes, the operating system will recognize the I/O device and will already have a software

interface (that is, a device driver) embedded in the operating system to converse with the I/O device. This means that we don't have to supply any Windows 98 code, any iMac code, any Linux code, or whatever to enable our I/O device to operate with these systems. A good point to note here is that USB devices can be used on all systems that support a USB port and these class drivers: An Intel PC keyboard operates on an iMac computer, and an iMac mouse operates on an Intel PC (aren't standards wonderful!).

If we are building a unique I/O device that the host operating system will not know about, we must supply a device driver, or *n* device drivers, one for each OS that we want to support. This is a lot of work, and discussion of writing a device driver is deferred until Chapter 10.

There are two special class codes: 0 indicates that the class code is defined in a later descriptor; 0FFH indicates a vendor-supplied class code.

All other codes describe a USB Specification Class Type—this list is constantly growing, and an on-line reference is kept on www.usb.org. I have provided more discussion about device classes within the examples that use them later in the book.

EP0 size defines the maximum number of bytes that can be sent or received by endpoint 0 in each data payload (the control endpoint that is always open). This is implementation-dependent and different values are used by different vendors (as we shall see in the next chapter).

To sell USB products in the open market, you must have a **Vendor ID**. These are issued by admin@usb.org for a small registration fee. For our examples which won't ship out of your lab we'll use 04242H, which is my Vendor ID. We choose whatever values are appropriate for **Product ID** and **Version#**.

The next three values are **indexes** into an array of strings. We'll use real values in our later examples, but for now we'll use 0 for the default values.

The final value of **Configurations** defines the number configurations that this device has. Let us assume for now that this is 1.

To recap enumeration step 7 (and 9), the PC host discovered that the device was a conforming I/O device that it should talk to. Any illegal values in the device descriptor would cause the PC host software to ignore the device.

Configuration Descriptor

In enumeration Step 9, the PC host requests a configuration descriptor. Figure 3-3 shows the minimum example—this is for a device that has one configuration with a single interface that uses only endpoint 0 to exchange data. Endpoint 0 is predeclared, and therefore it need not be redeclared.

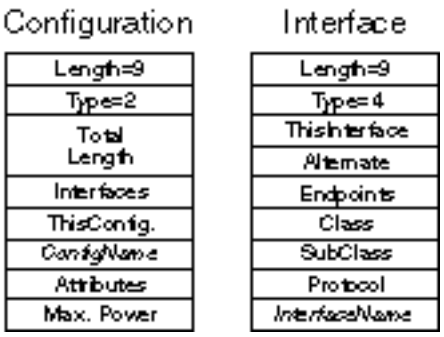


Figure 3-3. Descriptors for a minimal I/O device

A configuration descriptor has a fixed **length** and **type**: 9 and 2, respectively.

When the PC host requests the configuration descriptor, the device responds with all the descriptors that it uses to describe the functionality and operation of the device. These descriptors are concatenated together and sent as a single large block to the PC host. The **TotalLength** of this block is entered at byte offsets 2 and 3.

The number of **interfaces** supported by this configuration is entered at byte offset 4. A minimum I/O device will enter a 1 here.

A device with multiple configurations will have multiple configuration descriptors (we'll discuss this in the next section). Each configuration descriptor must have a unique identifier, called **ThisConfig**, and this is entered at byte offset 5.

The byte at offset 6 is an **index** into a string array. Since we are not using these yet, we enter a 0 for this entry.

Configuration **Attributes** is a bit-mapped field with bits 7, 6, and 5 defined as shown below and bits 4 through 0 reserved (set to 0):

- Bit 7 should be set to 1.
- Bit 6 set indicates that this device has its own power source.
- Bit 5 set indicates that this device can generate a remote wakeup.

We will use 50 in the **MaxPower** entry to indicate that we need only the 100 mA that was initially provided (values are in increments of 2mA). With this value, we would be considered a low-power device. We could request up to 500 mA from the bus, and some of the later examples will need this. Current requirements over 500 mA demand a self-powered device that includes a "wall-bug." A device can use both USB power and self-power. It is common to power the USB interface from the bus and use an external source for other device power. If the device power levels can be kept below 500 mA, then the device will not need an external power source. This will decrease the cost of the device and make it easier to use. It is worth the design effort to come in under the 500 mA limit. A device that needs to generate a REMOTE_WAKEUP will need external power to be able to generate this signal.

Interface Descriptor

A minimum device will have a single interface descriptor. The **length** and **type** are fixed at 9 and 4, respectively. We will enter 0 in the **Interface**, **Alternate**, and **Endpoints** fields, because they are not used in a minimum device.

The **Class**, **Subclass,** and **Protocol** entries are used to determine which software driver will be bound to this interface.

A PC host device driver is bound to this interface descriptor. A device that supports multiple interfaces will have a device driver bound to each interface. It is important to realize that the PC host software views USB as a large collection of SOFTWARE INTERFACES—the I/O device is a convenient hardware implementation and container for these software interfaces. There is a matching software driver for each interface descriptor. Multiple, similar interface descriptors can be supported by a single driver.

For our initial examples I will specify the HID class (=3). This means that two more descriptors need to be added to our minimal I/O device – while this is a little more added complexity now it will result in lower total effort for the project as we shall see later.

We use 0 for the **InterfaceName** because we are not yet using strings.

DEFINING A HUMAN INTERFACE DEVICE

Mice, keyboards, and game pads are good examples of HID devices. They react very slowly (when compared with a 700-MHz Pentium IV processor) and transfer only small amounts of information between the PC host and the I/O device. Many typical HID class devices include indicators, specialized displays, audio feedback, and force or tactile feedback. Therefore, the HID class definition includes support for various types of input and output directed toward the user.

A HID does not require a human interface! A HID class device supplies or consumes low amounts of data at infrequent times. If your device has a low data rate (below 64KB/s for a full-speed device or below 8KB/s for a low-speed device) then it could fit into the HID class.

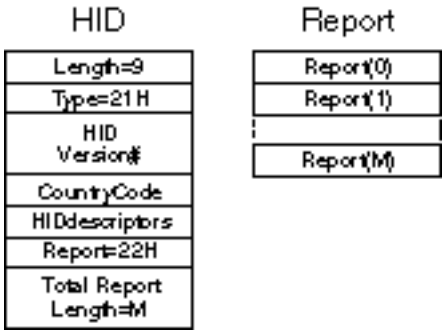
The HID descriptor would have been concatenated with other descriptors and supplied to the PC host following a Get_Configuration request. The HID device driver will read the Report descriptor later to discover exactly what kind of HID device is being defined.

Many of the initial examples in this book are human interface devices, so we need to slow down here to absorb all of the information in this section. Although HID operation sounds difficult at the first reading, it will turn out to be the easiest option to use to connect a unique I/O device to the PC host.

The **HID** class consists primarily of devices that people use to control the operation of computer systems. Typical examples include:

- Keyboards and pointing devices: mouse devices, trackballs, joysticks
- Front panel controls: knobs, switches, buttons, sliders
- Controls that might be found on devices such as telephones, VCR remote controls, games or simulation devices: data gloves, throttles, steering wheels, pedals
- Devices that may not require human interaction but provide data in a similar format to HID-class devices: bar code readers, thermometers, voltmeters

A Human Interface Device, or HID-class device, is identified by a class code of 3 in the interface descriptor. This entry will prompt the PC host to look for a HID Descriptor and a Report descriptor as shown in Figure 3-4.



<< Note last Report element should be Report(M-1) >>

Figure 3-4. HID and report descriptor formats

HID Descriptor

The first four entries of an HID descriptor have standard values: 9, 21H, 100H, and 0. If the HID device operates only in a single country, then **CountryCode** should be set appropriately (for more information, see Section 6.2.1 of USB Documents/HID Class Definition.pdf on the CD-ROM). A keyboard would be a good example of a country-specific I/O device.

The fifth entry, **HIDDescriptors**, contains a count of the number of HID class descriptors that follow. A single I/O device could have several HID interfaces (a keyboard with integrated trackball for example).

The sixth entry is fixed at x22H indicating that this interface contains a **report descriptor,** and the seventh entry contains the **TotalLength** of this report descriptor.

Report Descriptor

Hid devices communicate with the PC host using blocks of data called Reports. These blocks can be any length and the format of the data is specified using a report descriptor as described below.

If you are providing data to some pre-existing software, such as providing keyboard or mouse data to the operating system then the structure of this data is pre-defined and you must provide the data in the format that is expected by the receiving software. We'll do an example like this in Chapter 7.

If you are designing a unique device that interacts with unique PC host software then one of the first tasks is to define the format of the data that is exchanged on USB. Notice that we are defining the data format independantly of the I/O device hardware implementation. This logical representation allows the I/O device to be radically changed providing it maintains its Report interface. This structure gives hardware independence to the software, enabling both to be developed or improved on different schedules.

There are a great variety of HID devices so defining a report descriptor scheme that would encompass all possible devices was a large task.

The primary and underlying goals of the Report Descriptor are:

- Be as compact as possible to save device data space.
- Allow the software application to skip unknown information.
- Be extensible and robust.
- Support nesting and collections.
- Be self-describing to allow generic software applications.

A type of pseudocode has been generated that is able to generically describe any style and quantity of data (in Chapter 5, I'll describe an HID tool that does the work for you). The coding was chosen to be very compact and to make it easy for the PC host to parse and generate. As a result, it is a little difficult to read, as evidenced by the keyboard descriptors in Figure 3-5. I have added comments to help you decipher the report. This report generates a 1-byte OUTPUT buffer with 5 valid bits that is sent to the keyboard to illuminate the status LEDs; from the keyboard, the report receives a 8-byte INPUT buffer that can contains a modified byte, a reserved byte and up to six keystrokes.

```
Usage Page (Generic Desktop),   ;Use the Generic Desktop
        Usage (Keyboard),                   ;Start Keyboard collection
                Collection (Application),
                Usage Page (Key Codes),
                Usage Minimum (234),
                Usage Maximum (231),
                Logical Minimum (0),
                Logical Maximum (1),        ;Fields values from 0 to 1
                Report Count (8),
                Report Size (1),
                                            ;Add fields to the input report.
                Input (Data, Variable, Absolute),
                Report Count (5),           ;LED Report = 5 LEDs
                Report Size (1),            ;Size of each report item is 1 bit
                Usage Page (LEDs),
                Usage Minimum (1),
                Usage Maximum (5),
                Output (Data, Variable, Absolute),
                Report Count (3),
                Report Size (1),
                Output (Constant),          ;3 bits of Padding
                Report Count (6),           ;6 characters are buffered
                Report Size (8),            ;Characters are byte wide
                Logical Minimum (0),
                Logical Maximum (101),
                Usage Page (Key Codes),
                Usage Minimum (0),
                Usage Maximum (101),
                Input (Data, Array),        ;6 character buffer
                End Collection,
        End Collection
```

Figure 3-5. Commented keyboard report

A Report Generator tool is provided on the CD-ROM to help build custom reports. All reports used in our initial HID examples use the same buffer format: A byte-wide buffer of up to 64 entries can be sent to the I/O device, and the I/O device can supply a similar buffer. Since we are writing the software at both ends of the "cable" i.e. the application software and the device firmware then we can choose whatever buffer format is appropriate for each report. Each example will individually interpret the byte fields.

Choosing a Device Driver

The operating system device driver selection scheme gives you an extreme amount of control. I will use the Windows 98 scheme for illustration purposes - other operating systems have similar schemes. Windows uses all of the descriptor information that it has just read in to decide which device driver, or drivers, need to be loaded to support this newly attached device.

Windows first looks in the registry to determine if this device has ever been attached before – let us assume, for a moment, that it hasn't. Control is then passed to the Hardware Installation Wizard that first searches through every INF file in the Windows/inf directory tree looking for a match on the I/O device's VID and PID values.

A Windows INF file is a specially formated text file that defines the linkage between an I/O device and the device driver(s) required to enable it to be integrated into, and supported by, the operating system. We'll discuss the format in great detail in later chapters but, since out initial goal is to use a pre-existing driver for our examples, we will not provide a custom INF file!

If a VID/PID match is not found the wizard searches again for a match on the I/O device CLASS code. If none if found then the user is prompted to supply an INF file that the wizard can search. The wizard **must** find a match or it will not install the I/O device (every I/O device must have a corresponding device driver).

The wizard will record the match in the registry so that a driver can quickly be located at the next attach time or operating system reboot. In fact, several registry entries are made describing the device hardware, it's class, it's required drivers and other needed OS services.

A cautionary note for all developers*: during development it is easy to create "garbage" registry entries. Windows, of course, doesn't realize that they are garbage produced by errors during development and will use ANY matching registry entry when an I/O device is attached. A bad registry entry can slow down your project so make it a habit during your development cycle to "clean" the registry often (delete bad entries or overwrite with a know good registry). Microsoft designed the registry scheme to be easy for end-users; it is, but it does create problems for developers. Deleting matching VID/PID entries from the registry will force the hardware wizard to reinstall your device drivers and to create new registry entries.*

The initial examples in this book are all HID-class devices and will match with HIDDEV.SYS and therefore use the pre-installed device drivers. In other words, we don't need to supply an INF file.

MINIMUM I/O DEVICE

With multiple states to operate in and with control and data packets to respond to, we can agree that even a minimum USB I/O device is more than "a few PALs." Figure 3-9 shows a block diagram of the essential elements of a USB I/O device.

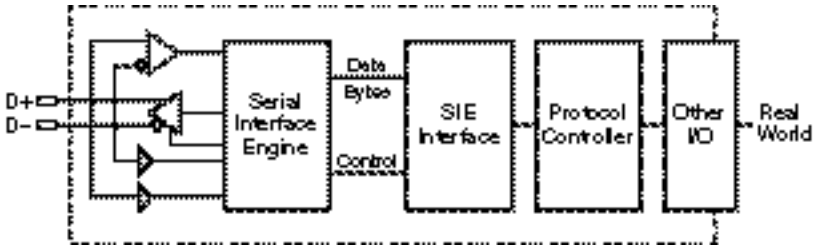


Figure 3-9. Essential elements of a USB I/O device

An essential part of a USB interface is the Serial Interface Engine, or SIE. The SIE receives bits from the USB transceiver, validates them, and provides valid bytes to the SIE interface. Similarly, bytes are received from the SIE interface and transmitted serially onto the USB bus.

The SIE contains intelligence to manage the USB bit-level protocol including the bit-stuffing algorithm. A vendor can include more intelligence in the SIE if desired or can pass raw data with possible error status to the protocol controller. The richness of the SIE interface also varies with different vendors—some supply the bytes from USB in FIFOs, some in memory; some supply error status flags and expect the protocol controller to deal with error conditions, while some error-check the incoming data, implement the USB handshake protocol, and only interrupt the protocol controller once clean, validated data has been received. In the next chapter we'll use a variety of USB components and will get first-hand experience of these implementation differences.

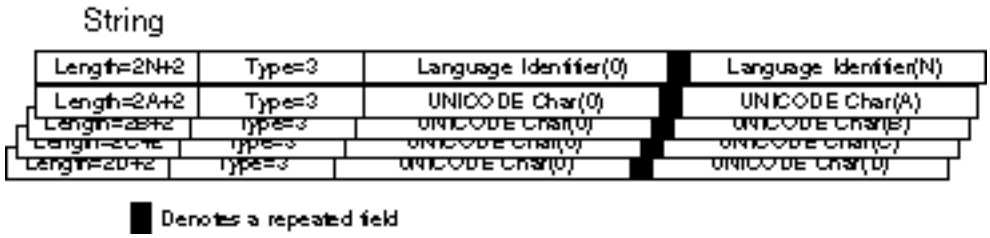
Enhancing the Minimum Device

This section expands on the basic descriptors described earlier in the chapter. We will edit the descriptors slowly so you can appreciate their building-block nature.

First we'll add a **Strings** descriptor. This has a variable length header followed by variable length string entries (Figure 3-10). The header consists of a **length** and **type** entry (2N+2 and 3) followed by an array (N) of **LanguageIdentifiers**.

A language identifier is ???

Each string entry consists of a **length** and **type** entry followed by a unicode **string**. A unicode string uses a word to represent each character and is not NULL-terminated. For more information on unicode, please refer to *The Unicode Standard, Worldwide Character Encoding*, produced by The Unicode Consortium and published by Addison-Wesley, Reading, Massachusetts, U.S.



<< Note all lengths should be (n-1) >>

Figure 3-10. Format of a strings descriptor

The order in which the strings are declared will define their **INDEX**—the indexes start from 1 because a 0 is used to define "no string." We can now use useful, human-readable strings in our I/O device and go back to our descriptors and back-fill the string index entries. These strings are helpful during the debug and enumeration phases because they allow to operating system to better identify the device (or it uses "unknown device," which is not very helpful).

Our basic descriptors used the pre-existing endpoint 0 for run-time data exchanges. This is adequate for some low-bandwidth applications, but endpoint 0 does not support interrupt, bulk or isochronous transfers. If we need one or more of these types of endpoints, we declare them using endpoint descriptors (Figure 3-11).

Figure 3-11. Format of an endpoint descriptor

The **length** and **type** of an endpoint descriptor are fixed at 7 and 5, respectively. If multiple endpoints are defined, each will need an endpoint descriptor that is uniquely identified by the **EndpointAddress** entry.

The endpoint address uses bits 3:0 to specify the endpoint number and bit 7 to specify if this is an IN endpoint (=1) or an OUT endpoint (=0). Bits 6:4 are reserved and set to 0.

The **Attributes** entry uses bits 1:0 to specify the endpoint type, control (=00), isochronous (=01), bulk (=10), or interrupt (=11). Bits 7:2 are reserved and set to 0.

The word at byte offset 4 specifies the **MaxPacketSize** that this endpoint can support.

The **Interval** entry is used by isochronous endpoints (must be set to 1) and by interrupt endpoints (set a value from 1 to 255) to specify the interval time, in milliseconds, of PC host polling.

To use a specific example see the interrupt endpoint descriptors used in Figure 3-12 to improve the performance of a HID device. Without these added endpoints the minimal HID device will accept and supply reports using the control commands SET_Report and GET_Report. We saw in the previous chapter that control transactions contained overheard.

The endpoints are declared as an Input (I/O device provides data to the PC host) and an OUTput (PC host provides data to the I/O device) endpoint.

PC host software can now schedule IN transactions to collect data and OUT transactions to provide data. The I/O device an NAK the IN request if data is not available and can NAK the OUT request if it does not have a buffer available. The overhead is lower and results in better bus utilization.

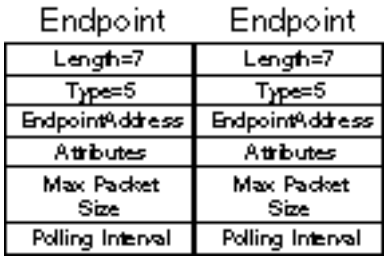


Figure 3-12. Adding two interrupt endpoints to a HID device

Alternate-Speed Descriptors

A high speed I/O device is capable of operating at high-speed and full-speed and has two sets of descriptors, one for each speed. The capabilities of the device may be different depending upon its operating speed so two additional descriptor types are used to enable the PC host to discover the capabilities at the other speed.

Figure 3-13 shows these two additional descriptors – the Device_Qualifier is similar to a Device_Descriptor and the Other_Speed_Configuration uses the same format as a Configuration_ Descriptor.

<<Paste from USB 2.0 Spec>>

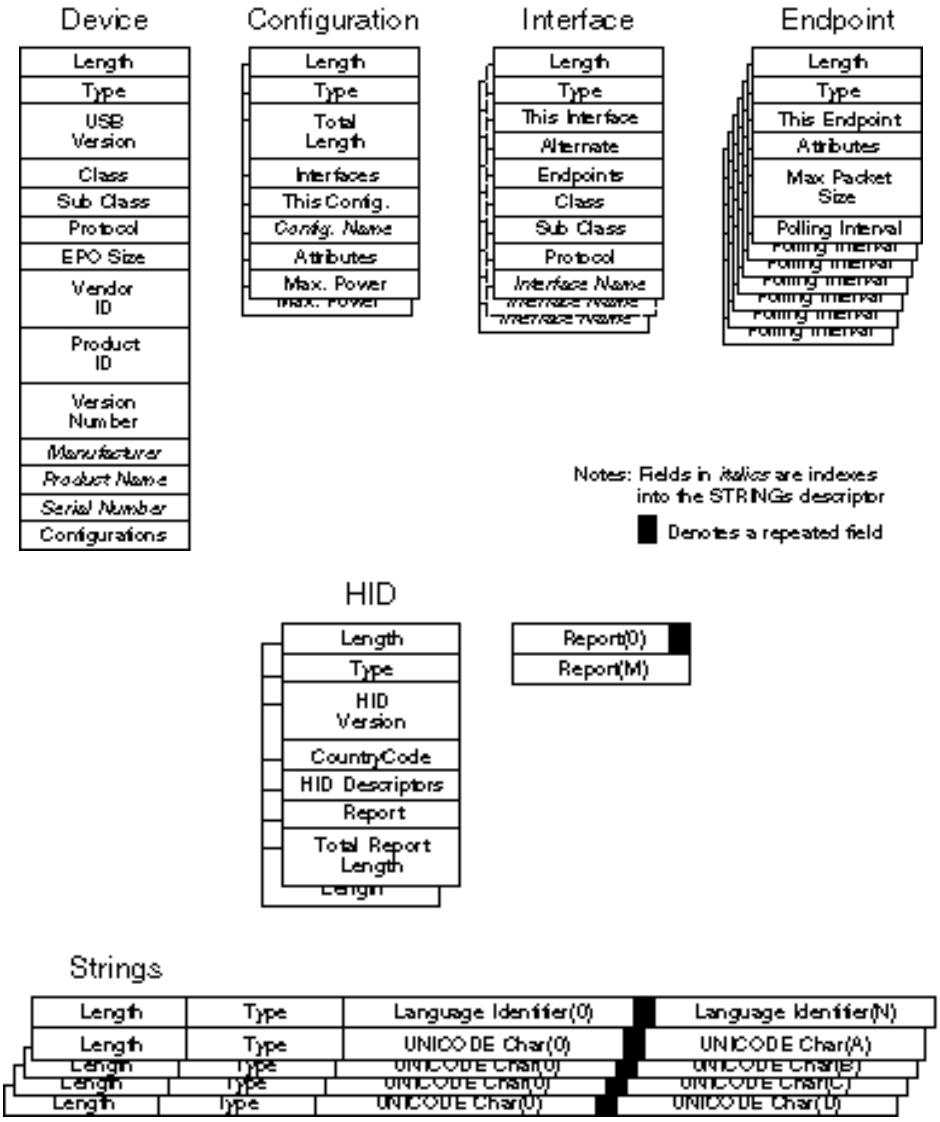
Figure 3-13 Alternate Speed Descriptors

Descriptors for a feature-rich I/O device.

Many devices have multiple interfaces and therefore multiple interface descriptors. Each interface descriptor specifies one or more endpoint descriptors required to implement this interface. Because these multiple interfaces are supported concurrently, each must specify exclusive use over its endpoints. That is, an endpoint cannot be referenced by more than one **active** interface descriptor.

An interface descriptor is **active** if it is specified in the **current configuration**. Some devices have more than one configuration, and only one configuration may be selected at one time. Each configuration may reference some of the interface descriptors of other configurations, but these become active only if contained in the currently selected configuration.

The flexible scheme results in a collection of descriptors (Figure 3-13). The total length of a configuration descriptor can be quite long: remember, this is the concatenation of all the descriptors except the device descriptor, the strings descriptor and any class descriptors (report in this example).



<< Note all lengths should be (n-1) >>

Figure 3-13. Descriptors for a feature-rich I/O device

ADDITIONAL I/O DEVICE RESPONSIBILITIES

We have already seen that the I/O device exists in various states—disconnected, attached, powered, etc. Figure 3-7 shows the complete state diagram that an I/O device must adhere to. A new state, called **suspended**, is important for correct I/O device operation and is introduced here.

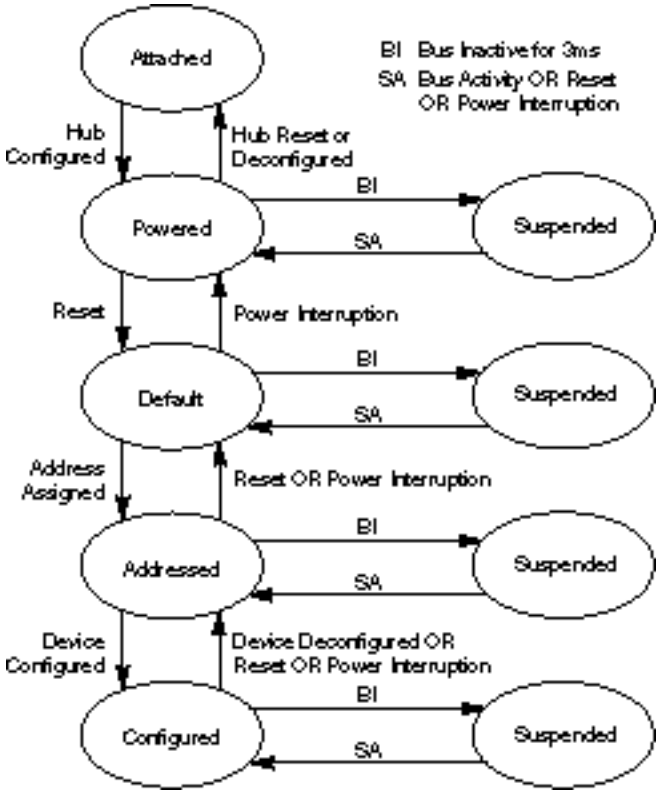


Figure 3-7. Required I/O device state diagram

If an I/O device detects no USB activity for 3 ms, the device is **required** to move to a low-power **suspended** state, in which it draws no more than 0.5 mA (average) from the bus. No bus activity for 3 ms means that the PC host has stopped sending SOF packets; this is a result of the PC host powering down. There's no point in keeping most I/O devices powered on if the PC host has powered down! Any activity on the bus will result in the I/O device returning from a suspend state into one of the active states.

It is possible for the I/O device itself to bring the PC out of its powered-down state. This capability, called **REMOTE_WAKEUP**, is the **only** time a signal is initiated by the I/O device. If the I/O device were a telephone, for example, it would want to wake up the PC if the phone rang. This capability must be predeclared in the device configuration descriptor so that the PC host knows to prepare for such an event.

The I/O device drives a remote wakeup signal (a K state onto the idle bus) to alert its local hub. Hubs propagate this signal up to the root hub, which informs the PC to wake up.

It should be noted that the PC host is allowed to send control requests even after the I/O device is in the configured state. Indeed, the PC host could send a Set_Configuration request specifying the zero configuration that moves the I/O device back to the addressed state where it stops responding to the real world.

Enumeration from the device point of view is straightforward (Figure 3-8). Requests are received from the PC host, and the I/O device must respond to them. The I/O device designer predeclares all of the descriptors and supplies the correct information to the PC host on request. We will work through several design examples in Chapter 7.

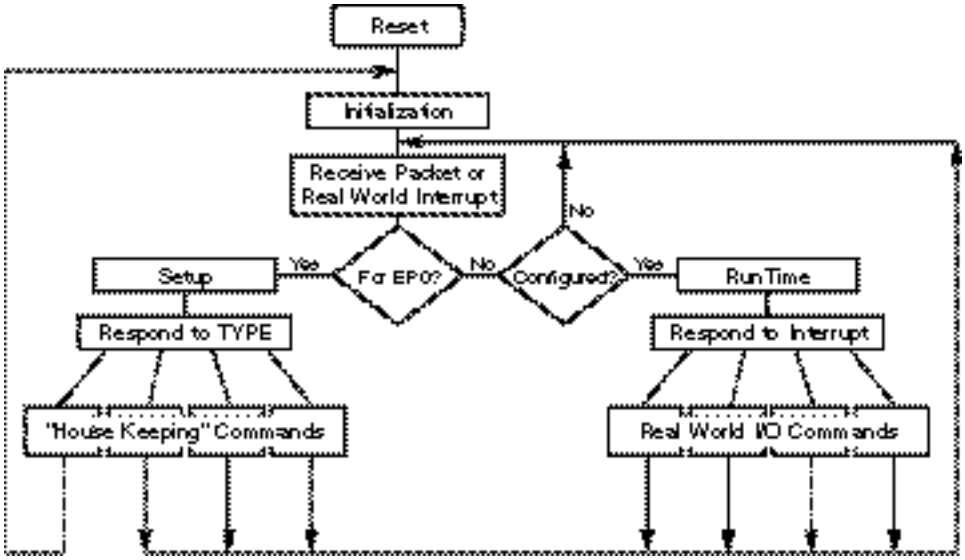


Figure 3-8. Typical software structure of an I/O device

CHAPTER SUMMARY

It has now been about 100 ms since the beginning of this chapter, and the PC host is ready to use the I/O device that we have added.

We saw how the enumeration phase was completed and learned the responsibilities of an I/O device. We stepped through the initialization and configuration information that was delivered to the PC host. The descriptors for a simple device are small, but the scheme allows for feature-rich devices that can have several configurations, each supporting multiple interfaces. An I/O device requires a USB transceiver, a Serial Interface Engine, and a protocol controller. This is more than "a few PALs," but low-end USB controllers may even be cheaper than these PALs. A wide range of hardware exists to engineer a USB I/O device and the next chapter presents a wide range of commercial devices—a device can be selected to match your design task.